Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC2343 – Arquitectura de Computadores

Ayudantía 1 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

Ayudante: Germán Leandro Contreras Sagredo (glcontreras@uc.cl)

Nota al lector

El título dice 'solución propuesta' por una razón bien sencilla: Estos ejercicios pueden tener más de un desarrollo correcto. Lo que se pretende hacer aquí es mostrar un camino a la solución, sin excluir la posibilidad de rutas alternativas igual de correctas.

Preguntas

1. a. **(I1 - II/2014)** Describa el valor decimal del número 0x94A6, si este se interpreta como binario con signo.

   Una forma rápida de expresar un número hexadecimal como uno binario, es transformando cada dígito de este a base binaria con 4 dígitos (recordando que $2^4 = 16$):

   - $9 = 1001_2$
   - $4 = 0100_2$
   - $A = 1010_2$
   - $6 = 0110_2$

   Luego, nuestro número en base binaria corresponde a 1001010010100110_2. Como este se interpreta como binario con signo, y el bit más significativo corresponde a un 1, utilizamos el complemento a 2 para obtener la representación correcta:

$$1001010010100110_2 = -C_2(1001010010100110_2) = -0110101101011010_2 = -27482$$

b. **(I1 - II/2011)** Dados $A = 45$ y $B = 57$, ¿cuál es el resultado, en binario, de la operación $A - B$?

Notemos que si queremos representar estos números en binario, y estamos realizando una operación que implica una resta, entonces necesariamente debemos considerar el bit de signo. Tenemos entonces que $A = 0101101_2$ y $B = 0111001_2$. Luego, restarle a un número positivo uno negativo es equivalente a sumarle su complemento a 2. Entonces:

$$-B = C_2(0111001_2) = 1000111_2$$

$$A - B = 0101101_2 + 1000111_2 = 1110100_2$$

Ahora, este número es negativo, por lo que si queremos su valor decimal correspondiente, realizamos nuevamente el complemento a 2:

$$A - B = -C_2(1110100_2) = -0001100_2 = -12$$

Finalmente, vemos que el número obtenido en binario es consistente con el resultado de la operación en base decimal.

c. Suponga que se tiene un total de 6 bits, usados para representar números positivos y negativos. Dados $A = 27$ y $B = 8$, ¿cuál es el resultado, en binario, de la operación $A + B$? ¿Por qué da este resultado?

Si se tiene un total de 6 bits, podemos representar sin problemas los A y B dados como números positivos. Tenemos entonces que $A = 011011_2$ y $B = 000100_2$. Luego, al realizar la operación:

$$A + B = 011011_2 + 001000_2 = 100011_2$$

Podemos ver que el resultado, utilizando representación binaria con signo, es negativo. El número, en base decimal, sería entonces:

$$100011_2 = -C_2(100011_2) = -011101_2 = -29$$

Esto sucede debido a que sobrepasamos nuestro poder de representación. Con 6 bits, el máximo número que podemos representar es $011111_2 = 31$, Si la suma nos da un resultado mayor a ese, al no ser capaces de representar dicho número, obtenemos un número incorrecto (pues la suma de dos números positivos no pueden dar como resultado uno negativo). Este resultado se conoce como **overflow**.

d. **(I1 - II/2014)** Si hay algún problema eléctrico, como un alza de voltaje, es muy fácil corromper datos almacenados en binario. Por ejemplo, el número 10 (1010b) puede transformarse en 14 (1110b), con tan solo modificar un bit. Describa una codificación binaria para los números 0 y 1, de manera que esta permita detectar y corregir errores de a lo sumo 1 bit, i.e., un bit de la codificación se ve alterado.

Una codificación binaria sencilla para realizar esto, es codificar cada bit como 3 bits de sí mismo, es decir:

$$f(x) = \begin{cases} 111, & x = 1 \\ 000, & x = 0 \end{cases}$$

De esta forma, si tuvieramos el número 0101, este resultaría $f(0101) = 000111000111$. Si uno de los bits se corrompe, basta ir revisando de a 3 dígitos el número, y detenernos cuando no veamos que los 3 bits coinciden para encontrar el espacio que fue corrompido. El bit que predomine corresponderá al número original.

Notar que esto no habría funcionado para la siguiente codificación:

$$g(x) = \begin{cases} 11, & x = 1 \\ 00, & x = 0 \end{cases}$$

Esto, ya que si bien podemos identificar el lugar de corrupción revisando de a 2 dígitos, no podemos saber cuál correspondía al números original (por ejemplo, 10 pudo haber sido 11 o 00, no lo sabemos con dicha codificación).

e. **(I1 - I/2017)** ¿Cuál es la cantidad máxima de pixeles que puede tener una imagen en blanco y negro de 1KB no comprimida? Asuma que cada pixel solo almacena su valor de color y que el archivo de la imagen solo almacena pixeles, por lo que no es necesario considerar el encabezado.

Como nos interesa saber si un color es blanco o negro, tenemos dos posibilidades de color, por lo que basta con un bit para representarlo. Luego:

$$1KB = 1024B = 1024 * 2^3 b = 8192b$$

Es decir, necesitamos 8192 bits para la imagen, lo que equivale a 8192 pixeles.

2. a. Implemente, utilizando solo las compuertas lógicas `AND`, `OR` y `NOT`, el conectivo binario condicional ($\rightarrow$), que está definido por la siguiente tabla de verdad:

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

Se puede implementar fácilmente si recordamos que:

$$A \rightarrow B \Leftrightarrow \neg A \vee B$$

Esto se puede corroborar con la equivalencia existente en sus tablas de verdad.
Ahora, la fórmula lógica se puede implementar de la siguiente forma con conectivos lógicos:

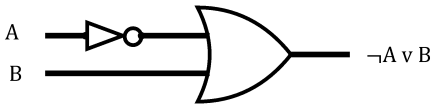


Figura 1: Circuito resultante.

b. **(Apuntes - Operaciones aritméticas y lógicas)** Implemente un circuito 2 bit Multiplier, que realice la multiplicación entre dos valores de 2 bits
Para ver esto de forma sencilla, primero vemos cómo implementar una multiplicación entre dos números de un bit:

A	B	$A * B$
0	0	0
0	1	0
1	0	0
1	1	1

Es fácil ver que la multiplicación la podemos realizar a partir de la compuerta lógica `AND`. Ahora, como nos piden una multiplicación entre dos números de dos bits, tenemos que seguir el método de multiplicación tradicional. Para cumplir este objetivo es importante recordar la composición de los *half-adders*, en la que la suma entre dos números se representa por la compuerta `XOR`, mientras que el *carry* resultante se hace con la compuerta `AND`[1].

[1] **¿Por qué?** Porque si ambos bits de entrada son iguales a 1, entonces sabemos que la suma resultará en 0, "sobrando" una unidad (como lo vemos en la suma tradicional).

Sean A y B dos números de dos bits de la forma A_1A_0 y B_1B_0. Si queremos obtener el número $C = A * B$, seguimos el siguiente procedimiento:

- Multiplicamos el bit menos significativo de B por los dos bits de A. De esta forma, tendremos A_0 `AND` $B_0 = C_0$, y A_1 `AND` $B_0 = C_1^{'}$.
- Multiplicamos ahora el bit más significativo de B por los dos bits de A. De esta forma, tendremos A_0 `AND` $B_1 = C_1^{''}$ y A_1 `AND` $B_1 = C_2^{'}$
- El bit menos significativo de nuestro resultado será C_0. Luego, el bit siguiente se obtiene de la siguiente forma: $C_1 = C_1^{'}$ `XOR` $C_1^{''}$ (tal como lo hacemos con la suma). Como esto nos puede generar un carry, tomamos $C_2^{''} = C_1^{'}$ `AND` $C_1^{''}$
- Ahora, el siguiente bit lo conseguimos como la suma entre el producto de los bits más significativos de cada número, sumado al carry anterior: $C_2 = C_2^{'}$ `XOR` $C_2^{''}$.
- Finalmente, como nuestro número puede tener máximo 4 bits (11 * 11 = 1001), tomamos el bit más significativo del resultado como el carry de la última suma: $C_3 = C_2^{'}$ `AND` $C_2^{''}$.

Finalmente, nuestro circuito queda de la siguiente forma:

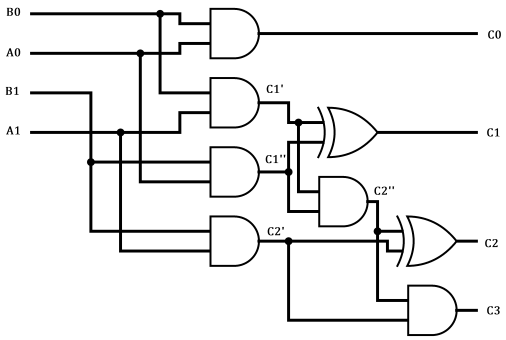


Figura 2: Resultado del circuito descrito.

Notar que este diagrama se puede simplificar haciendo uso de *half-adders* y *full-adders*.

c. **(I1 - I/2017)** Construya un circuito que permita detectar la ocurrencia de *oveflow* al sumar o restar dos números enteros de 8 bits en una ALU.

Para construir este circuito es necesario identificar los cuatro casos en los que se produce *overflow*, ya sea por la adición o sustracción de dos números A, B:

- **Caso 1:** $A \geq 0, B \geq 0, A + B = M < 0$
- **Caso 2:** $A < 0, B < 0, A + B = M \geq 0$
- **Caso 3:** $A \geq 0, B < 0, A - B = M < 0$
- **Caso 4:** $A < 0, B \geq 0, A - B = M \geq 0$

Para cada caso formaremos un circuito distinto, de forma que se puedan conectar todos al final. Para identificar el signo, utilizamos el bit más significativo de cada número (A, B, S) y lo conectamos en un puerto `AND` que reciba, además, el bit esperado. Luego, para identificar la operación, hacemos uso de otro puerto `AND` que recibe dos entradas: El número esperado del comando ejecutado en la ALU (según sea el caso) y el número de operación recibido (que denotaremos por S). Entonces, los circuitos generados son los siguientes:

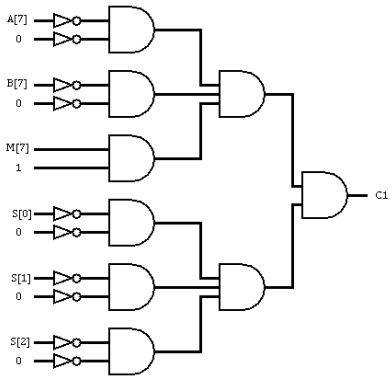


Figura 3: Caso 1.

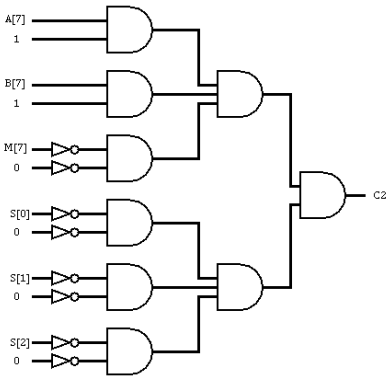


Figura 4: Caso 2.

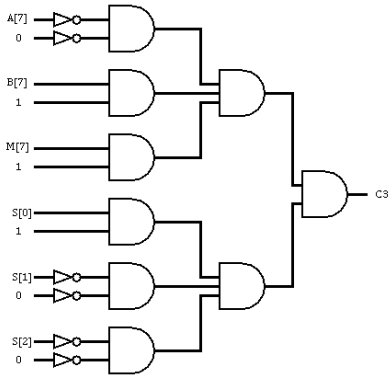


Figura 5: Caso 3.

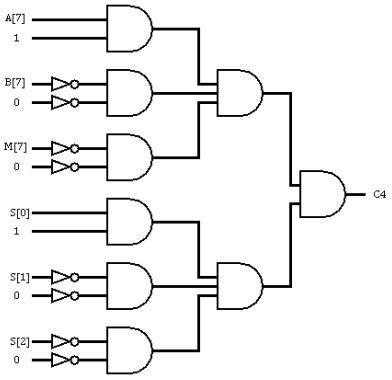


Figura 6: Caso 4.

Finalmente, conectamos los resultados a un puerto `OR` para obtener el bit que indica si se generó o no *overflow* (el que llamamos O):

Figura 7: Conexión final entre los cuatro circuitos antes descritos.

d. **(I1 - I/2012)** Diseñe un De-Multiplexor con bus de datos de 1 bit y bus de control de 2 bits.

A diferencia de los multiplexores, los de-multiplexores se encargan de transmitir la señal que reciben solo a una de sus salidas, la que se especifica mediante una señal de selección. A partir de esta descripción, el diagrama de un Demux es el siguiente:

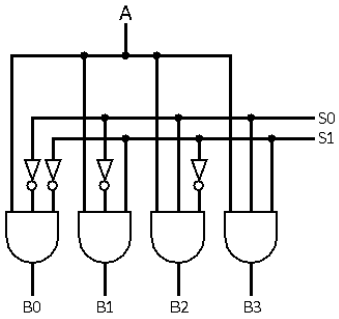


Figura 8: De-Multiplexor con entrada de 1 bit y bus de control de 2 bits.

e. **(I1 - I/2017)** ¿Qué número entero es generado al realizar cuatro operaciones **shift right** seguidas de cinco operaciones **rotate left** a un registro de 8 bits que inicialmente almacena el número entero 79?

Para interpretar bien el resultado, consideramos la representación binaria para números enteros con signo (pues, por enunciado, sabemos que debemos considerar números negativos).

Tenemos que $79 = 01001111$. Ahora, iremos listando los números resultantes:

- `shift right`: $00100111 \rightarrow 39$
- `shift right`: $00010011 \rightarrow 19$
- `shift right`: $00001001 \rightarrow 9$
- `shift right`: $00000100 \rightarrow 4$
- `rotate left`: $00001000 \rightarrow 8$
- `rotate left`: $00010000 \rightarrow 16$
- `rotate left`: $00100000 \rightarrow 32$
- `rotate left`: $01000000 \rightarrow 64$
- `rotate left`: $10000000 \rightarrow$ -128

Notemos que en esta secuencia el uso de `rotate left` no difiere de `shift left`, ya que no hubo ningún bit entre medio que pudiera ser conservado. Si se hiciera un `rotate left` más:

- `rotate left`: $00000001 \rightarrow 1$

Aquí sí se aprecia la diferencia, donde el bit más significativo ahora se vuelve el menor en vez de perderse, como es en el caso de `shift left`.

3. a. **(I1 - II/2016)** Modifique un *latch* tipo RS agregando una señal de control C, tal que los cambios en el estado del *latch* solo se realicen cuando $C = 1$.
Una posible implementación consiste en el siguiente circuito:

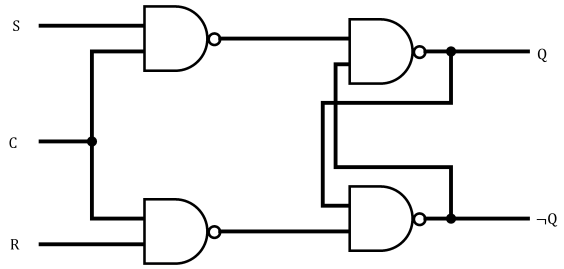


Figura 9: Primera opción.

La diferencia con el *latch* original es que se invierten las señales S y R para mantener sus funciones en el circuito (pues se intercambian con la compuerta `NAND` y queremos que la señal R y S almacenen un 0 y un 1, respectivamente). Por otra parte, la combinación de señales indefinida correspondía a R = 0 y S = 0, pero ahora corresponde a R = 1 y S = 1 (lo que no es relevante siempre que se indique). La tabla, entonces, queda así:

R	S	C	Q	$\overline{Q}$
X	X	0	Q	$\overline{Q}$
0	0	1	Q	$\overline{Q}$
0	1	1	1	0
1	0	1	0	1
1	1	1	?	?

Cuadro 1: Tabla de verdad asociada a la primera opción.

Sin embargo, el siguiente circuito también sirve:

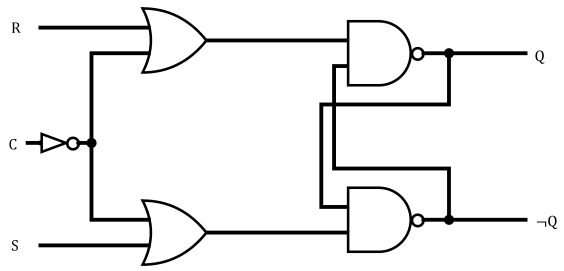


Figura 10: Segunda opción.

En este también se cumple con el objetivo de controlar los cambios de estado con la señal C, pero además no es necesario invertir el orden de las señales R y S (pues cumplen sus funciones como corresponde). Por otra parte, se mantiene la combinación que indefine los estados del circuito. Finalmente, la tabla queda de la siguiente forma:

R	S	C	Q	$\overline{Q}$
X	X	0	Q	$\overline{Q}$
0	0	1	?	?
0	1	1	1	0
1	0	1	0	1
1	1	1	Q	$\overline{Q}$

Cuadro 2: Tabla de verdad asociada a la segunda opción.

b. **(I1 - I/2017)** En la siguiente figura, si la frecuencia del *clock* que entra al *flip-flop* FF0 es F Hz, ¿cuál es la frecuencia del *clock* del *flip-flop* FFN?

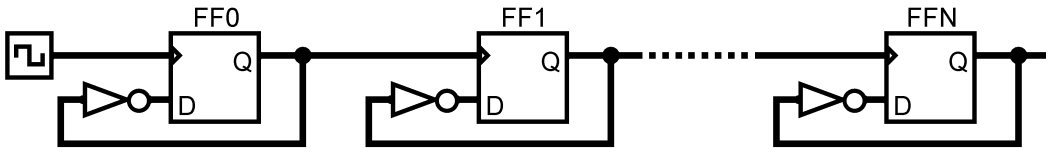


Figura 11: Secuencia de *flip-flops*, donde el *clock* de uno es la señal de estado del que lo antecede, salvo para el primero.

La parte clave de esta pregunta es notar la señal de clock que recibe FF1. Podemos ver que en vez de ser la misma señal del principio, es el estado Q del *flip-flop* FF0. Ahora, pensemos cómo varía la señal Q_{FF0} (i.e. la señal Q del *flip-flop* FF0). Si la frecuencia de dicho *flip-flop* es F, entonces a Q_{FF0} le toma un tiempo $\frac{1}{\text{F}}$ cambiar su valor (independiente si es de 0 a 1 o de 1 a 0). El hecho de que la entrada D reciba como valor la negación del estado Q_{FF0} es lo que permite simular una frecuencia (alternando sus valores). Finalmente, como Q_{FF0} se demora $\frac{1}{\text{F}}$ segundos en cambiar su valor y otros $\frac{1}{\text{F}}$ segundos en volver al inicial, su periodo es de $\frac{1}{\text{F}} + \frac{1}{\text{F}} = \frac{2}{\text{F}}$, lo que implica que su frecuencia será $\frac{\text{F}}{2}$.

Si seguimos esta idea de forma análoga para la señal de clock que recibe FF2, veremos que a Q_{FF1} le toma $\frac{2}{\text{F}}$ segundos cambiar su valor y $\frac{2}{\text{F}}$ segundos volver al original, obteniendo un periodo de $\frac{2}{\text{F}} + \frac{2}{\text{F}} = \frac{4}{\text{F}}$ y una frecuencia de $\frac{\text{F}}{4}$.

A partir de lo anterior, obtenemos la siguiente relación de recurrencia:

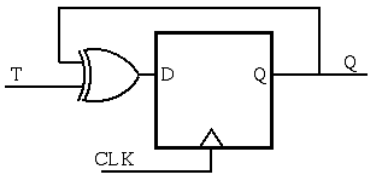
$$f_{\text{FF}}(i) = \begin{cases} \frac{f_{\text{FF}}(i-1)}{2}, & i > 0 \\ F, & i = 0 \end{cases}$$

Donde $f_{\text{FF}}(i)$ representa la frecuencia del *flip-flop* i. Finalmente, podemos ver que:

$$f_{\text{FF}}(N) = \frac{f_{\text{FF}}(N-1)}{2} = \frac{f_{\text{FF}}(N-2)}{4} = ... = \text{F} \prod_{i=0}^{N-1} \frac{1}{2} = \frac{\text{F}}{2^N}$$

c. **(I1 - I/2013)** Diseñe usando compuertas lógicas y *flip-flops* D, un *flip-flop* T. El comportamiento de este *flip-flop* consiste en invertir el valor de su salida Q si su señal de entrada T está en 1 y la señal de control C pasa de 0 a 1 (flanco de subida). En cualquier otro caso, la salida Q se mantiene igual.

Una posible solución se presenta en el siguiente diagrama:



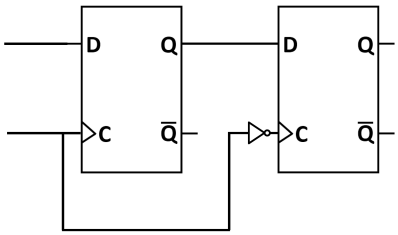
De partida, vemos que la salida del *flip-flop* D correspondiente a la señal Q se conecta a una compuerta `XOR`, la que recibe además la señal T. Esto genera la siguiente tabla de casos:

T	Q	T `XOR` Q
0	0	0
0	1	1
1	0	1
1	1	0

Así vemos que, efectivamente, para $T = 1$ el resultado de la compuerta es el inverso de la señal Q. Se obtiene finalmente la siguiente tabla para el *flip-flop* T:

T	D	Q
0	Q	Q
0	Q	Q
1	$\overline{Q}$	$\overline{Q}$
1	$\overline{Q}$	$\overline{Q}$

d. **(I1 - II/2012)** Implemente mediante compuertas lógicas, elementos de control y latches, un *flip-flop* tipo D que funcione con flanco de bajada.
Lo que se busca, básicamente, es lo siguiente:



En el *flip-flop* tipo D tradicional, se tiene una negación antes de la entrada de la señal de control para ambos *latches*. Esto permite que solo uno de estos circuitos almacene un valor a la vez (según la señal C), habilitando cambios en el contenido guardado en la estructura completa solo en los flancos de subida. Ahora, notemos que al eliminar la primera negación, el primer *latch* (llamémoslo de entrada) solo permitirá el almacenamiento de un dato cuando $C = 1$. En cambio, el segundo *latch* (que llamaremos de salida) solo guarda el estado para $C = 0$. De esta forma, cuando existe la transición C: $1 \rightarrow 0$ (i.e., el flanco de bajada) el *latch* de entrada permite almacenar el dato y lo transfiere al *latch* de salida, siendo este último capaz de guardarlo correctamente. Esto se logra antes de que el *latch* de entrada vuelva a cambiar su valor (al habilitarse nuevamente por C), lo que permite que el *flip-flop* D pueda guardar la señal en cada flanco de bajada.

e. **(I1 - I/2012)** Implemente mediante compuertas lógicas y *flip-flops* tipo D, el registro de la figura, con señales de control (C), carga (*Load*) y *reset* (*Reset*), que funciona con flanco de subida.

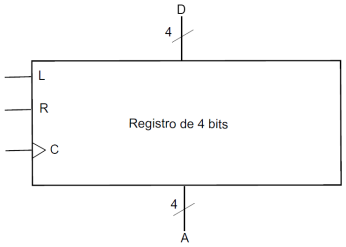
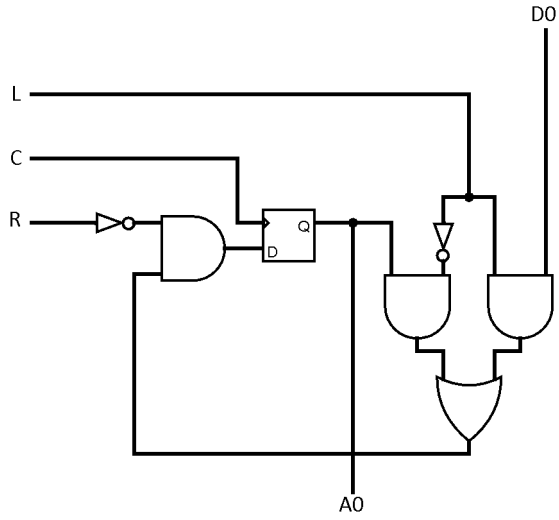


Figura 12: Registro de 4 bits que funciona con una señal de control, carga y *reset*.

Antes de ver la figura, es necesario entender bien la funcionalidad de cada señal:

- ***Load*:** Habilita el almacenamiento del valor que está recibiendo el registro.
- **C:** Permite que existan cambios en el registro solo en los flancos de subida. Corresponde al clock.
- ***Reset*:** *Resetea* el valor almacenado en 0.

El siguiente diagrama representa el circuito que permite lograr la funcionalidad antes explicada:



Es importante notar que esto solo ilustra el almacenamiento del bit menos significativo en el registro. No obstante, el almacenamiento del resto de los bits es el mismo.

Analicemos ahora las partes relevantes del circuito:

- **L:** Si la señal L está activa, se debe permitir el paso del nuevo bit para que se dirija a la entrada del *flip-flop* D. A su vez, debe evitar que este pueda almacenar el antiguo valor contenido por este. Por esta razón, se utiliza la señal L en dos compuertas `AND`: En la de la derecha, se preocupa de dejar pasar al dato que se busca ingresar, mientras que en la de la izquierda se bloquea el paso del bit original (por eso se utiliza una negación en L). Es importante notar que si L está inactiva, los papeles se invierten: Se bloquea el paso del bit de entrada y se permite el flujo del valor original del *flip-flop*. Finalmente, se escoge el resultado a partir de la compuerta `OR` (el que haya sido negado quedará descartado al ser igual a 0).
- **C:** Al corresponder a la señal del *clock*, se conecta directamente al *flip-flop* D para permitir el cambio de valores almacenados solo en los flancos de subida.
- **R:** Si esta señal está activa, se debe cambiar el valor almacenado en el *flip-flop* a 0, sin importar el resto de las señales. Esto se logra a partir de una compuerta `AND` entre la negación de la señal R y el resultado del paso de bits explicado en el primer punto. Si R está activo, su negación será igual a 0, lo que implicará que el resultado de la compuerta será 0 independiente de lo que ingrese en la otra entrada, logrando resetear el *flip-flop*. En cambio, si R está inactivo, su negación será igual a 1, lo que hace que el valor a almacenar finalmente sea equivalente al resultado obtenido por la señal L.

f. **(I1 - II/2012)** ¿Cuántas direcciones tiene una memoria RAM de 4.5 KB que utiliza palabras de 3 bytes? (1KB = 1024 bytes).

Recordemos que el espacio de memoria utilizado por una RAM está definido como el producto entre el número de direcciones y el tamaño de la palabra almacenada en cada dirección. Luego, nos basta con despejar la siguiente ecuación (hacemos el cálculo usando los bytes como unidad):

$$\text{Número de direcciones} * 3[bytes] = 4{,}5 * 1024[bytes] \rightarrow \text{Número de direcciones} = 1536$$

g. Suponga que se tiene una matriz almacenada en la dirección de memoria 0x0A. Esta posee un total de 4 filas y 5 columnas. Si se sabe que en una dirección de memoria se puede almacenar 1 byte, y la matriz almacena en cada celda un dato de 2 bytes, ¿cuál es la dirección del dato que se encuentra en la tercera columna de la segunda fila de la matriz? Asuma que se utiliza la convención de filas.

Para este ejercicio, basta con recordar la fórmula para la obtención de datos dentro de una matriz (según la convención de filas):

$$dir(matriz[i][j]) = dir(matriz) + i * sizeof(matriz[i][j]) * M + j * sizeof(matriz[i][j])$$

Donde:

- $dir(matriz[i][j])$ es la dirección que buscamos.
- $dir(matriz)$ es la dirección donde se comienza a almacenar la matriz.
- $sizeof(matriz[i][j])$ es el tamaño utilizado por cada celda en la matriz.
- M es la cantidad de columnas.
- $[i][j]$ es la fila y columna correspondiente a la dirección buscada.

Finalmente, reemplazando, tenemos que:

$$dir(matriz[1][2]) = 0x0A + 1 * 2 * 5 + 2 * 2 = 0x0A + 14 = 0x18$$

Notar que si bien parte utilizando la dirección 0x18, al ser este dato de dos bytes y la capacidad por dirección de un byte, tenemos que utiliza las direcciones 0x18 y 0x19 para almacenar el dato completo.